JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 31 No. 6, pp. 1-15, June 2026
https://doi.org/10.9708/jksci.2026.31.06.001

HAL: An Encoding Method for Efficient Log Data Pipelines

Min-gi Lee*, Choong-hee Cho**

*Student, Division of Computer Science and Engineering, Sahmyook University, Seoul, Korea
**Professor, Division of Computer Science and Engineering, Sahmyook University, Seoul, Korea

[Abstract]

In recent large-scale service environments, data pipelines are widely used to collect and transmit web logs in real time. In this process, even when serialization methods such as Avro are applied, repetitive values remain in the data, leading to increased network transmission costs. To address this limitation, this study proposes HAL (Hybrid Analytics Log), a lightweight encoding method that can be integrated into existing systems without significant structural modifications. The proposed approach replaces frequently repeated string combinations with integer identifiers, while preserving unmatched data in its original form to ensure lossless processing. Experimental results using Nginx access logs show that the proposed method reduces Kafka message size by up to 16.2%, Kafka payload bandwidth by 17%, and data ingestion time by 24.6%, without any significant degradation in total processing time or throughput. In addition, the Parquet storage size is further reduced by 4.8% in the final storage stage. These results demonstrate that the proposed approach effectively improves both transmission and storage efficiency through low-cost computational operations.

▸ **Key words**: Data pipeline, log processing, serialization, data compression, network efficiency, Kafka

[요 약]

최근 대규모 서비스 환경에서는 웹 로그와 같은 데이터를 실시간으로 수집 및 전송하기 위한 데이터 파이프라인이 널리 사용되고 있다. 이 과정에서 Avro와 같은 직렬화 방식을 적용하더라도 반복 값이 그대로 포함되어 네트워크 전송 비용이 증가하는 한계가 있다. 본 연구에서는 이러한 반복 데이터를 효율적으로 처리하기 위해 기존 구조에 큰 변경 없이 적용 가능한 경량 인코딩 방식인 HAL(Hybrid Analytics Log)을 제안한다. 제안 방식은 자주 반복되는 문자열 조합을 정수형 식별자로 치환하고 매칭되지 않는 데이터는 원본 그대로 유지하여 무손실 특성을 보장한다. 제안 방식은 Nginx 액세스 로그를 대상으로 한 실험에서 Kafka 메시지 크기를 최대 16.2%, Kafka 페이로드 대역폭을 17%, 데이터 적재 시간을 24.6%까지 감소시키면서도 전체 처리 시간과 처리량에는 유의미한 영향을 주지 않는 것으로 나타났다. 또한, 최종 저장 단계에서도 Parquet 데이터 크기를 추가로 4.8% 절감하여 저비용 CPU 연산을 통해 네트워크 및 저장 효율을 동시에 개선할 수 있음을 확인하였다.

▸ **주제어**: 데이터 파이프라인, 로그 처리, 직렬화, 데이터 압축, 네트워크 효율성, Kafka

• First Author: Min-gi Lee, Corresponding Author: Choong-hee Cho
*Min-gi Lee (mk8048@naver.com), Division of Computer Science and Engineering, Sahmyook University
**Choong-hee Cho (cch@syu.ac.kr), Division of Computer Science and Engineering, Sahmyook University
• Received: 2026. 04. 23, Revised: 2026. 04. 30, Accepted: 2026. 05. 21.

http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

I. Introduction

클라우드 네이티브 아키텍처와 마이크로서비스 아키텍처(Microservices Architecture, MSA)의 확산으로 인해 시스템 상태를 추적하고 분석하기 위한 로그 및 모니터링 파이프라인 구축은 현대 산업 환경에서 필수 요소로 자리 잡았다 [1]. 모놀리식 아키텍처(Monolithic Architecture)와 달리, MSA 환경은 다수의 독립된 컨테이너로 구성되며 분산되고 동적인 특성을 지니기 때문에 대규모 로그 데이터가 생성되고, 이에 따라 분석 및 모니터링의 복잡도가 증가하고 있다 [2]. 이러한 로그 데이터를 수집하고 집계하기 위해 Logstash, Fluentd와 같은 수집 에이전트가 널리 사용되고 있으나 [3], 클라우드 네이티브 환경이 고도화됨에 따라 기존 데이터 수집 구조는 확장성과 실시간 처리 측면에서 한계를 보이고 있다 [4]. 본 연구는 이 중 로그 수집 경로에서의 전송 효율성 문제에 초점을 맞춘다. 본 연구는 다양한 로그 유형 가운데 웹 서비스의 ingress 계층에서 생성되는 Nginx access log 기반 수집 경로를 1차 검증 대상으로 한정한다. Nginx와 같은 reverse proxy access log는 HTTP 메서드, 엔드포인트, 상태 코드, 프로토콜, 콘텐츠 유형 등 필드 의미가 명확하고 반복성이 높은 값을 포함하므로, Avro 직렬화 이후에도 남는 값 수준 중복 문제를 관찰하기에 적합하다. 따라서 본 연구의 목적은 모든 로그 유형에 대해 반복적이고 의미가 명확한 필드 조합이 존재하는 대표적인 웹 access log 수집 경로에서 직렬화 이전 전처리 인코딩의 효과를 검증하는 것이다.

방대한 로그 처리를 위해 Apache Kafka와 같은 분산 메시지 브로커가 중추 인프라로 널리 활용되고 있으며 산업계에서는 스키마 레지스트리와 연동된 Avro 바이너리 직렬화를 통해 Kafka로 데이터를 전송하는 방식이 사실상 표준으로 자리 잡고 있다 [5][6]. Avro 직렬화는 JSON 대비 필드명의 반복 전송을 제거하여 데이터 크기를 줄이는 효과가 있다. 그러나 로그 데이터에서 높은 빈도로 반복되는 값의 종류가 제한적인 저카디널리티(Low Cardinality) 문자열 값, 예를 들어 HTTP 메서드, 엔드포인트, 상태 코드 등은 Avro 인코딩 과정에서 제거되지 않으며 [7][8], 필드명 중복이 제거되더라도 값 수준의 중복성은 직렬화된 데이터에 그대로 남게 된다. 이러한 데이터가 Kafka 클러스터로 지속적으로 유입될 경우, 브로커 노드의 네트워크 대역폭 포화와 디스크 쓰기 병목이 발생할 수 있으며 [5] 이는 수집기 측에서 처리 속도를 초과하는 데이터 유입으로 인해 입력을 지연시키거나 제한하는 현상을 유발하여 전체 수집 구조의 처리량 저하로 이어질 수 있다[9][10].

이러한 전송 계층의 I/O 병목에 대해 기존 연구들은 컬럼형 스토리지의 배치 분석 최적화 [11][12], Avro 및 Protobuf 기반의 스키마 의존적 키 중복 제거 [6], Kafka 레벨의 범용 블록 압축 [13] 등 상이한 층위에서의 접근을 취해 왔다. 그러나 이러한 접근들은 실시간 수집 경로에 직접 적용하기 어렵거나, 값 수준의 중복성을 충분히 제거하지 못하며, 추가적인 CPU 오버헤드를 유발하는 한계를 가진다. 따라서 스트리밍 수집 경로에서 로그 데이터의 값 수준 중복을 줄이기 위한 접근은 아직 충분히 고려되지 않았다.

본 연구에서는 이러한 한계를 보완하기 위해 JSON에서 Avro를 거쳐 Parquet으로 이어지는 수집 파이프라인의 효율을 향상시키는 하이브리드 로그 형식인 HAL(Hybrid Analytics Log)을 제안한다. Hybrid는 두 가지 표현 방식을 결합한다는 의미를 가진다. 반복적으로 등장하는 주요 필드 조합은 사전 기반 정수 식별자로 치환하여 전송 크기를 줄이고, 사전에 정의되지 않은 조합은 예외 값 배열에 원본 값을 보존하여 무손실 복원을 보장한다. 또한 레코드 구조 측면에서는 분석에 자주 사용되는 핵심 필드와 활용 빈도가 낮은 부가 필드를 분리함으로써 조회 효율성과 스키마 확장성을 함께 고려한다. HAL은 단순한 바이트 단위 처리 방식과 달리, HTTP 메서드, 엔드포인트, 상태 코드와 같이 로그 각 필드가 가지는 의미를 기반으로 반복되는 값의 조합을 사전에 정의하고 이를 간결한 형태로 변환하는 방식으로 동작한다. 제안 방식은 기존 수집 구조를 전면적으로 변경하지 않고, 직렬화 이전 단계에서 적용되는 경량 전처리 계층으로 설계되었다. 이를 통해 반복적으로 등장하는 주요 필드 조합은 빠른 탐색을 통해 단일 식별 값으로 변환되며, 사전에 정의되지 않은 경우에도 원본 정보를 유지할 수 있도록 처리된다. 또한 활용 빈도가 낮은 부가 정보는 별도로 분리하여 저장함으로써 스키마 변경 없이도 새로운 필드를 유연하게 수용할 수 있도록 하였다. 이러한 구조는 직렬화 과정에서 처리해야 할 데이터의 양을 감소시키며, 최종 저장 단계에서도 데이터 압축 효율을 향상시키는 데 기여한다.

본 논문의 주요 기여는 다음과 같다.

첫째, 기존 Avro 기반 직렬화가 해결하지 못하는 로그 데이터의 값 수준 중복 문제를 완화하기 위해 스트리밍 수집 단계에서 반복되는 필드 조합을 사전에 변환하는 스키마 인지형 사전(dictionary) 기반 전처리 기법을 제안하였다. 이를 통해 기존 방식이 처리하지 못했던 중복 구조를 효과적으로 제거할 수 있음을 보였다.

둘째, 제안 방식은 Kafka 기반 로그 수집 파이프라인에 구조 변경 없이 적용 가능한 전처리 계층으로 설계되어 기존 시스템과의 호환성을 유지하면서도 전송 및 저장 효율을 동시에 개선할 수 있음을 실험적으로 검증하였다.

셋째, 제안 방식은 추가적인 복잡한 압축 연산이나 모델 기반 처리 없이 단순한 연산만으로 네트워크 사용량과 데이터 크기를 줄이는 구조를 가지며, CPU 자원을 활용하여 네트워크 비용을 절감하는 효율적인 자원 활용 전략을 제시한다.

넷째, 다양한 필드 조합과 매칭 비율에 따른 성능 변화를 체계적으로 분석하여 전송 효율과 처리 비용 간의 관계를 정량적으로 규명하고 실제 시스템 적용을 위한 설계 기준을 제시하였다.

본 논문의 나머지 구성은 다음과 같다. 2장에서는 로그 압축 및 경량화 분야의 선행 연구를 검토하고 본 연구가 겨냥하는 연구 공백을 제시한다. 3장에서는 HAL 프레임워크의 파이프라인 통합 개요, 설계 원칙, 핵심 구성 요소, Avro 스키마 설계를 상술한다. 4장에서는 실험 환경과 데이터셋, 변인 설정 및 측정 방법을 포함한 실험 설계를 기술하고, 5장에서는 그 결과를 통계적 유의성과 함께 분석한다. 마지막으로 6장에서 결론과 한계점 및 향후 연구 방향을 제시한다.

II. Related work

로그 데이터의 저장 비용과 처리 부하를 줄이기 위한 연구는 처리 단위에 따라 파일 또는 청크 단위의 사후 압축 방식과 스트림 또는 레코드 단위의 실시간 압축 방식으로 구분할 수 있다.

1. File and Chunk Level Compression

이 범주의 연구들은 로그 파일 전체 또는 일정 크기의 청크를 대상으로 반복 패턴을 식별하고 이를 압축하는 데 초점을 둔다. 이러한 접근은 로그 데이터의 구조를 분석하여 반복되는 템플릿과 변동 파라미터를 분리하고, 이를 효율적으로 인코딩함으로써 높은 압축률을 달성하는 것을 목표로 한다.

Logzip [14]은 비정형 텍스트 로그에서 반복되는 템플릿을 반복 군집화 기법을 통해 자동으로 추출하고, 이를 정수형 인덱스로 치환한 뒤 범용 압축기를 적용하는 방식을 제안하였다. LogReducer [15]는 파라미터 간 상관관계를 활용한 인코딩과 타임스탬프 차분 기법을 결합하여 이를 확장하였으며, LogShrink [16]는 데이터를 컬럼 단위로 재구성하여 동질적인 데이터 분포를 형성함으로써 압축 효율을 향상시켰다. LogFold [17]는 서로 다른 템플릿 간에도 공유되는 패턴을 식별하여 추가적인 압축 가능성을 확보하였고, LogPrism [18]은 구조 추출과 변수 인코딩을 통합하여 기존 방식의 한계를 보완하고자 하였다. 또한 LogBlock [19]은 실제 시스템에서 사용되는 소형 블록 단위에서 기존 기법의 성능 저하를 분석하였으며, Denum [20]은 템플릿 파싱 없이 숫자 토큰을 활용한 차분 인코딩 기법을 제안하였다.

이와 같은 기법들은 공통적으로 높은 압축률을 달성할 수 있다는 장점을 가지지만, 로그 데이터가 일정 수준 이상 축적된 이후에 수행되는 사후 처리 방식을 전제로 한다. 따라서 이벤트가 연속적으로 유입되는 실시간 스트리밍 수집 환경에 직접 적용하기에는 처리 지연과 시스템 구조 측면에서 한계를 가진다.

2. Stream and Record Level Compression

스트리밍 환경에서 발생하는 로그 데이터를 실시간으로 압축하기 위한 연구도 진행되고 있다. 이러한 접근은 데이터가 생성되는 즉시 압축을 수행함으로써 전송 비용을 줄이고, 저장 및 처리 효율을 개선하는 것을 목표로 한다.

Cowic [21]은 로그 스트림을 컬럼 단위로 분리하고 각 컬럼을 독립적으로 압축함으로써 로그 스트림 분석에서 압축 효율과 컬럼별 접근성을 개선하는 방식을 제안하였다. LogLite [22]는 사전 정의된 규칙이나 학습 과정 없이 슬라이딩 윈도우 내에서 유사 로그를 탐색하고, XOR 연산과 바이트 정렬 기반 기법을 결합한 경량 압축 알고리즘을 제시하였다. 이러한 접근은 로그 스트림 환경에서의 실시간 압축 가능성을 보여주었다는 점에서 의미가 있다.

그러나 이러한 기법들은 고처리량 환경에서 구조적인 제약을 가진다. 학습 기반 방식은 로그 패턴 변화에 따라 재학습이 필요하여 운영 비용이 증가할 수 있으며, 슬라이딩 윈도우 기반 탐색은 윈도우 크기에 비례하는 비교 연산을 요구하므로 CPU 병목으로 이어질 가능성이 있다. 또한 XOR 기반 압축 방식은 이전 로그에 의존하는 순차적 복원을 요구하므로, 소비자 측에서 임의 접근이 어렵고 컬럼 기반 저장 구조와의 호환성에도 제약이 따른다.

3. Summary and Research Gap

본 연구는 기존 로그 압축 및 경량화 기법과 다음과 같은 차별점을 가진다. 기존 연구들이 주로 로그 수집 이후의 압축 또는 스트림 내 일반적인 패턴 유사성에 기반한

압축에 초점을 둔 반면, 본 연구는 로그가 직렬화되기 이전 단계에서 반복되는 필드 값을 직접 변환하는 접근을 취한다. 이를 통해 기존 방식에서 제거되지 않는 값 수준의 중복을 구조적으로 감소시킬 수 있다. 또한 기존 기법들이 추가적인 압축 연산이나 모델 기반 처리에 의존하는 것과 달리, 본 연구는 사전에 정의된 필드 조합에 대한 단순 조회 연산만으로 데이터를 변환함으로써 연산 오버헤드를 최소화한다. 더 나아가 제안 방식은 특정 시스템에 종속되지 않고, Avro 직렬화와 Parquet 저장으로 이어지는 기존 수집 파이프라인에 직접 적용 가능하도록 설계되어 실용성을 확보하였다. HAL은 바이트열 수준의 범용 압축과 달리, 로그 레코드가 가지는 필드 의미와 값의 반복 구조를 직렬화 이전 단계에서 활용한다. 일반적인 압축 방식이 메시지 배치나 바이트열에서 반복 패턴을 탐색하는 데 초점을 두는 반면, HAL은 HTTP 메서드, 엔드포인트, 상태 코드와 같이 의미가 명확하고 반복성이 높은 필드 조합을 정수 식별자로 표현하여 직렬화 대상 데이터의 구조를 단순화한다. 이러한 점에서 HAL은 전송 또는 저장 단계의 압축 설정과 별개의 계층에서 동작하는 경량 전처리 인코딩으로 볼 수 있다. 이를 기반으로 본 연구에서는 JSON 구조화 로그를 입력으로 사용하는 Avro 기반 수집 파이프라인을 대상으로, 반복되는 필드 조합을 직렬화 이전 단계에서 처리하는 경량 전처리 기법을 제안하고, 통제된 실험 환경에서 Avro-only 기준 구조 대비 전송 및 저장 효율 개선 효과를 정량적으로 검증한다.

III. Proposed Framework: HAL

1. Pipeline Integration Overview

본 장에서는 HAL의 적용 위치와 내부 동작 방식을 설명한다. HAL은 기존 로그 수집 파이프라인을 대체하는 별도 시스템이 아니라, 직렬화가 수행되기 직전의 수집 단계에서 반복 값을 간결한 식별 값으로 변환하는 전처리 계층이다. 따라서 Kafka 브로커, Kafka Connect 기반 저장 커넥터, Parquet 저장 구조와 같은 기존 인프라 구성 요소를 변경하지 않고도 적용할 수 있다.

본 연구의 대상 파이프라인은 Fig. 1에 나타낸 바와 같이 세 단계로 구성된다. 첫 번째 단계에서는 Logstash가 애플리케이션으로부터 JSON 형태의 원본 로그를 수집한다. 이후 필터 단계에서 JSON 파싱, 타임스탬프 정규화, 필드 정제와 같은 기본 전처리를 수행한 뒤, 스키마 레지스트리에 등록된 Avro 스키마에 따라 바이너리 데이터로 직렬화한다. 즉, 수집, 기본 전처리, 직렬화 과정이 단일 Logstash 프로세스 안에서 수행된다.

두 번째 단계에서는 직렬화된 메시지가 Apache Kafka 토픽에 적재되어 전송된다. Kafka는 고처리량 로그 전송에 적합하지만, 메시지 크기가 증가할수록 브로커의 네트워크 대역폭 사용량과 디스크 쓰기 부하가 함께 증가한다. 세 번째 단계에서는 S3 Sink Connector가 Kafka 토픽의 메시지를 소비하고, Snappy 압축이 적용된 Parquet 형식으로 오브젝트 스토리지에 저장한다.

1장에서 제기한 병목은 주로 첫 번째 단계와 두 번째 단계 사이에서 발생한다. Avro 직렬화는 필드명 반복을 줄일 수 있으나, HTTP 메서드, 엔드포인트, 상태 코드처럼 값의 종류는 제한적이지만 빈번하게 반복되는 문자열 값은 그대로 남는다. 이러한 값들이 포함된 메시지가 지속적으로 Kafka에 유입되면 메시지당 전송 바이트 수가 증가하고, 브로커 노드의 네트워크 대역폭과 디스크 쓰기 부하가 커진다. 이 부하는 다시 수집기 측에서 입력을 지연시키거나 제한하는 현상으로 이어질 수 있으며 전체 수집 구조의 처리량을 저하시킬 수 있다.

HAL의 적용 지점은 Logstash 내부에서 기본 전처리가 완료된 직후이자 Avro 직렬화가 수행되기 직전이다. 이 위치에서 HAL은 반복적으로 등장하는 주요 필드 조합을 사전에 정의된 정수형 식별 값으로 변환한다. 그 결과 Avro가 직렬화해야 하는 데이터의 양이 줄어들고, Kafka로 전송되는 메시지 크기도 감소한다. 또한 최종 저장 단계에서는 반복되는 값이 더 정돈된 형태로 전달되므로 Parquet의 컬럼 기반 압축이 안정적으로 동작할 수 있는 입력 구조를 제공한다.

2. Design Principles and Core Components

앞서 정의한 하이브리드 특성에 따라 HAL의 설계는 네 가지 원칙을 따른다. 첫째, 반복되는 값의 조합을 직렬화 이전에 변환한다. 이는 Avro가 제거하지 못하는 값 수준의 중복을 줄이기 위한 것으로, 저장 단계에서 수행되는 Parquet의 사전 인코딩과는 적용 시점이 다르다. 둘째, 사전에 정의되지 않은 값이 유입되더라도 원본 정보를 보존한다. 이를 통해 새로운 요청 패턴이나 예외적인 로그가 발생하더라도 데이터 손실 없이 수집을 지속할 수 있다. 셋째, 각 레코드는 다른 레코드에 의존하지 않고 독립적으로 복원될 수 있어야 한다. 넷째, 분석에 자주 사용되는 필드와 활용 빈도가 낮은 부가 필드를 구분하여, 조회 효율과 스키마 확장성을 함께 고려한다.

Table 1. Combo table example (5-field)

combo_id	method	endpoint	status	protocol	content_type
1	GET	/	200	HTTP/2.0	text/html; charset=utf-8
2	GET	/search	200	HTTP/2.0	application/json
3	GET	/product	200	HTTP/2.0	application/json
4	GET	/products	200	HTTP/2.0	application/json
5	GET	/categories	200	HTTP/2.0	application/json

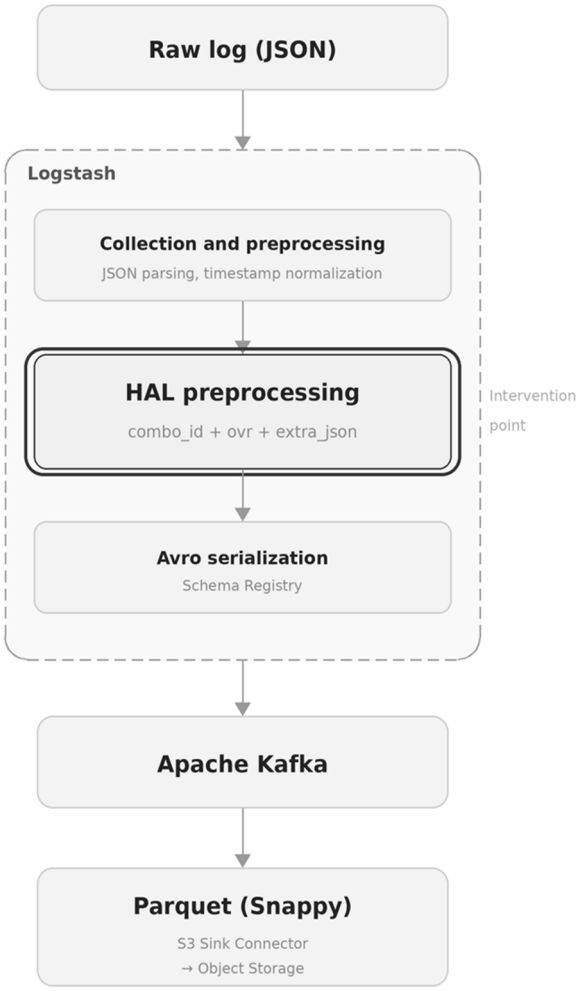

Fig. 1. HAL pipeline architecture overview

이러한 원칙에 따라 HAL 레코드는 핵심 필드, 조합 식별자, 예외 값 배열, 부가 필드 그룹의 네 요소로 구성된다. 핵심 필드는 모든 레코드에서 공통적으로 사용되며, 분석과 인덱싱에 자주 활용되는 필드이다. 본 연구에서는 스키마 버전, 조합 식별자, 타임스탬프, 응답 시간, 사용자 식별자, 세션 식별자를 핵심 필드로 구성하였다. 이 필드들은 레코드의 최상위에 배치되어 후속 분석 단계에서 별도의 중첩 구조 해석 없이 접근할 수 있다.

조합 식별자는 자주 반복되는 문자열 조합을 하나의 정수 값으로 표현하기 위한 필드이다. 이하에서는 구현상의 명확성을 위해 이 값을 combo_id로 표기한다. 예를 들어 HTTP 메서드, 엔드포인트, 상태 코드, 프로토콜, 응답 콘텐츠 유형이 동일한 조합으로 반복되는 경우, 해당 문자열들을 매번 전송하는 대신 하나의 정수형 식별자로 치환한다. Table 1은 5개 필드로 구성된 조합 테이블의 예를 보여준다.

인코딩 과정은 다음과 같다. Logstash의 필터 계층은 유입된 로그 레코드에서 조합 대상 필드 값을 추출하고, 이를 하나의 조회 키로 구성한다. 이후 해시 맵으로 구현된 조합 테이블에서 해당 키를 탐색한다. 매칭에 성공하면 대응되는 정수형 식별자를 combo_id에 기록하고, 원본 문자열 필드는 레코드에서 제거한다. 이 과정에서 여러 개의 가변 길이 문자열이 하나의 정수 값으로 치환되므로 직렬화 대상 데이터가 감소한다. 조합 테이블에 포함되는 필드 수는 설계자가 정하는 파라미터이며, 필드 수가 증가할수록 레코드당 절감 가능한 문자열 바이트 수는 증가하지만 매칭 비율은 낮아질 수 있다. 운영 환경에서 조합 테이블은 고정된 수작업 목록으로만 구성될 필요는 없으며 일정 관측 구간의 로그 샘플에서 필드 조합의 빈도와 예상 절감 바이트 수를 기준으로 생성할 수 있다. 조합 후보 필드는 값의 종류가 제한적이고 반복 빈도가 높으며 문자열 길이가 충분하여 정수 식별자 치환에 따른 이득이 큰 항목을 우선 선택한다. 이후 상위 N개의 조합을 등록하고, 운영 중 매칭 비율이 임계값 이하로 낮아지면 새로운 테이블 버전을 생성하여 배포할 수 있다. 이때 사전에 없는 조합은 ovr 배열에 원본 값으로 보존되므로 테이블 갱신 지연이 데이터 손실로 이어지지는 않는다. 등록 조합 수를 T, 조합 필드 수를 k라고 하면 레코드당 키 구성 비용은 O(k), 해시 맵 기반 평균 조회 비용은 O(1), 테이블 저장 비용은 O(T)로 볼 수 있다. 따라서 실제 운영에서는 테이블 크기 N, 매칭 비율, 레코드당 평균 절감 바이트 수를 함께 고려하여 테이블 갱신 주기와 필드 조합을 결정하는 것이 바람직하다.

조합 테이블에 등록되지 않은 값의 조합이 유입되는 경우에는 예외 값 배열을 사용한다. 이하에서는 이 배열을 ovr로 표기한다. 매칭 실패 시 combo_id에는 특수 값인 0을 기록하고, 원본 필드 값들은 조합 테이블의 필드 순서와 동일한 순서로 ovr 배열에 저장한다. 예를 들어 조합 테이블의 필드 순서가 method, endpoint, status,

```
{  "schema_id": 500,
  "combo_id": 4,
  "ts": 1766032740000,
  "rt_ms": 399,
  "uid": null,
  "sid": "c2e4e5f6-...",
  "ovr": [],
  "extra_json": {
    "request": "GET /products HTTP/2.0",
    "remote_addr": "43.135.145.77",
    "http_user_agent": "Mozilla/5.0 ..."
  }
}
```

(a) Combo hit (combo_id = 4)

```
{  "schema_id": 500,
  "combo_id": 0,
  "ts": 1766032740000,
  "rt_ms": 3002,
  "uid": null,
  "sid": "88a2e2aa-...",
  "ovr": [
    "GET", "/categories",
    "499", "HTTP/2.0",
    "application/json"
  ],
  "extra_json": {
    "request": "GET /categories HTTP/2.0",
    "remote_addr": "61.109.236.35",
    "http_user_agent": "Mozilla/5.0 ..."
  }
}
```

(b) Combo miss (combo_id = 0)

Fig. 2. HAL encoding examples: (a) Combo hit, (b) Combo miss

protocol, content_type이라면, ovr 배열의 첫 번째 값은 method, 두 번째 값은 endpoint에 대응된다. 이 방식은 필드명을 반복 저장하지 않으면서도 각 값의 위치와 의미를 명확히 보존한다. 매칭에 성공한 경우에는 ovr 배열이 비어 있으므로 추가적인 데이터 증가가 발생하지 않는다.

활용 빈도가 낮은 부가 필드는 별도의 그룹으로 분리한다. 이하에서는 이 그룹을 extra_json으로 표기한다. 본 연구에서는 원본 요청 문자열, 질의 매개변수, 클라이언트 주소, 참조 주소, 상품 식별자, 호스트, 응답 바이트 수, 상위 서버 응답 시간, 사용자 에이전트 등을 부가 필드로 구성하였다. 이러한 필드는 분석에 항상 사용되지는 않지만 디버깅이나 원본 추적에 필요할 수 있으므로 제거하지 않고 별도 그룹에 보존한다. 이 구조는 최상위 스키마가 과도하게 복잡해지는 것을 방지하면서도 신규 필드를 유연하게 수용할 수 있게 한다.

Fig. 2는 HAL 인코딩 결과를 매칭 성공과 매칭 실패 두 경우로 보여준다. Fig. 2(a)에서는 HTTP 메서드, 엔드포인트, 상태 코드와 같이 반복적으로 등장하는 문자열 필드가 단일 정수형 식별자로 대체되어 레코드 구조가 단순화되고 페이로드 크기가 감소하며, 예외 값 배열은 비어 있는 상태로 유지된다. 반면 Fig. 2(b)에서는 사전 정의된 조합과 일치하지 않는 경우로, 기본 식별자와 함께 배열에 원본 값이 저장되어 데이터가 손실 없이 보존된다. 이를 통해 HAL은 일반적인 반복 패턴에 대해서는 효율적인 표현을 제공하면서도, 예외적인 입력에 대해서는 원본 복원을 보장한다.

HAL로 인코딩된 레코드는 별도의 전용 복원 프로그램 없이 관계형 질의 방식으로 원본 구조에 가깝게 복원할 수 있다. combo_id가 0이 아닌 경우에는 조합 테이블과 조인하여 대응되는 원본 필드 값을 참조한다. combo_id가 0인 경우에는 ovr 배열의 인덱스 순서에 따라 각 필드 값을 추출한다. 핵심 필드와 부가 필드는 combo_id와 관계없이 항상 존재하므로 별도의 분기 없이 선택할 수 있다.

이러한 복원 과정은 식별자와 필드 조합 간의 일대일 대응, 배열 순서의 고정, 그리고 레코드 간 독립성에 의해 보장된다. 따라서 HAL은 스트리밍 환경에서도 레코드 단위의 독립성을 유지하며, 컬럼 기반 저장 구조에서도 필요한 필드만 선택적으로 조회할 수 있다.

3. Avro Schema Design

HAL 레코드를 Kafka로 전송하기 위해서는 스키마 레지스트리에 등록할 Avro 스키마가 필요하다.

Fig. 3은 본 연구에서 사용한 HAL 스키마 구조를 보여주며, 상위 레코드 내부에 핵심 필드, 예외 값 배열, 그리고 부가 필드 그룹이 계층적으로 구성된 형태를 확인할 수 있다. 상단 필드에서 볼 수 있듯이, schema_id와 combo_id는 정수형 타입으로 정의되어 있으며 이는 각각 스키마 버전과 필드 조합을 식별하기 위한 값이다. ts는 밀리초 단위의 Unix epoch를 저장하기 위해 long 타입으로 선언되었고, rt_ms는 응답 시간을 나타내는 정수형 필드로 구성된다. uid와 sid는 Fig. 3에서와 같이 null을 허용하는 문자열 타입으로 정의되어 값이 존재하지 않는 경우에도 직렬화 오류 없이 처리될 수 있도록 하였다. 또한 timestamp, request_time, user_id, session_id와 같은 필드는 각각 ts, rt_ms, uid, sid로 축약되어 스키마의 간결성과 저장 효율을 동시에 고려하였다. ovr 필드는 문자열 배열 형태로 정의되어 있으며 이는 Fig. 2(b)에서와 같이 매칭 실패 시 원본 필드 값을 순서대로 저장하기 위한 구조이다. 이 배열은 가변 길이를 가지며 HTTP 상태 코드와 같은 값도 문자열로 통일하여 저장함으로써 타입 변환 과정에서 발생할 수 있는 오류를 방지하고 복원 과정을 단순화하였다. 매칭 성공 시에는 해당 배열이 빈 배열로 유

지되므로 일반적인 경우 추가적인 데이터 증가가 거의 발생하지 않는다. extra_json 필드는 별도의 레코드 타입으로 정의되어 있으며 request, query_params 등 부가 정보를 포함하는 구조로 구성된다. 이와 같이 부가 필드를 별도의 레코드로 분리함으로써 상위 스키마의 구조를 단순하게 유지하면서도 다양한 필드를 유연하게 수용할 수 있다. 각 내부 필드는 null을 허용하는 문자열 타입으로 선언되어 일부 필드가 누락된 경우에도 동일한 스키마 내에서 처리될 수 있도록 하였다. 부가 필드 그룹의 표현 방식으로는 문자열, 맵, 레코드의 세 가지 방식을 고려할 수 있다. 문자열 방식은 JSON 형태로 직렬화되어 하나의 필드에 저장되므로 유연성은 높지만, Parquet 저장 시 단일 열로 처리되어 컬럼 단위 압축의 이점을 활용하기 어렵다. 맵 방식 역시 중첩 구조로 저장되어 필드별 반복 값에 대한 압축 효율이 제한된다. 반면 Fig. 3과 같이 레코드 타입을 사용할 경우, 각 필드가 독립적인 컬럼으로 분리되어 저장되므로 사전 인코딩 및 반복 길이 인코딩과 같은 컬럼 기반 압축 기법을 효과적으로 적용할 수 있다.

```
HAL Record {
  schema_id : int
  combo_id  : int
  ts        : long
  rt_ms     : int
  uid       : string (nullable)
  sid       : string (nullable)
  ovr : array<string>
  extra_json : record {
    request       : string (nullable)
    query_params  : string (nullable)
    ...
  }
}
```

Fig. 3. Avro schema for HAL

IV. Experimental Design

1. Experimental Infrastructure

본 장에서는 HAL의 효과를 검증하기 위한 실험 환경, 데이터셋, 변인 설정, 측정 방법을 설명한다. 실험의 목적은 제안 방식이 기존 수집 파이프라인의 구조를 유지하면서 메시지 크기, 네트워크 대역폭 사용량, 적재 시간, 최종 저장 크기를 얼마나 개선하는지 확인하는 것이다. 또한 전처리로 인해 발생하는 연산 비용이 전체 처리 시간과 처리량에 미치는 영향도 함께 측정하였다.

실험 환경은 KakaoCloud kr-central-2 리전의 동일 가상 사설 클라우드에 배치된 세 대의 가상 머신으로 구성하였다. 모든 노드는 AMD EPYC 7643 48-Core Processor 기반이며 운영체제는 Ubuntu 22.04.4 LTS로 통일하였다. 노드들이 동일한 가상 사설 클라우드 안에서 통신하므로, 외부 네트워크 지연으로 인한 변동을 최소화할 수 있다. 각 노드의 역할 및 하드웨어 사양은 Table 2에, 소프트웨어 구성과 버전 정보는 Table 3에 각각 정리하였다.

Table 2. Experimental node configuration

Node	Role	vCPU	RAM
Log Collection Node	Logstash (Producer)	2	8 GB
Messaging Node	Schema Registry, Kafka	4	16 GB
Storage Node	Kafka Connect (Consumer)	8	32 GB

Table 3. Software configuration

Node	Version
Logstash	7.17.29
Apache Kafka	3.7.1
Kafka Connect	3.7.1
Confluent S3 Sink Connector	12.0.0
Confluent Schema Registry	7.5.12
Java (Logstash JDK)	OpenJDK 11.0.30
Object Storage	KakaoCloud (kr-central-2)

2. Dataset and Pipeline Workload

실험 데이터셋은 온라인 쇼핑몰 서비스를 모사한 Flask 애플리케이션과 Nginx 리버스 프록시 환경에서 수집한 액세스 로그로 구성하였다. 본 모사 환경은 reverse proxy, application, database로 이어지는 일반적인 웹 서비스 요청 경로를 단순화한 구조이다. Nginx는 외부 HTTP 요청을 수신하는 ingress 및 reverse proxy로 동작하며 요청을 Flask 애플리케이션으로 전달하고, 요청 메서드, URI, 상태 코드, 프로토콜, 콘텐츠 유형, 응답 바이트 수, 응답 시간 등을 JSON access log로 기록한다. Flask 애플리케이션은 상품 조회, 검색, 카테고리 조회와 같은 읽기 중심 요청과 장바구니, 주문과 같은 쓰기 요청을 처리하며, 각 요청은 MySQL 데이터베이스의 조회 또는 쓰기 연산을 수반하도록 구성하였다. 최종 데이터셋은 총 849,000건의 로그 레코드로 구성되며, 각 로그 레코드는 Table 4에 제시한 18개 필드를 포함한다.

Table 4. Nginx access log field composition

Category	Field	Description
Core flat fields	timestamp	Request timestamp
	session_id	Session identifier
	user_id	User identifier
	request_time	Response time (sec)
Combo target fields	method	HTTP method
	endpoint	Request endpoint
	status	HTTP status code
	server_protocol	Protocol version
	sent_http_ content_type	Response content type
Extra fields	request	Raw request string
	remote_addr	Client IP
	http_referer	Referrer URL
	query_params	Query parameters
	product_id	Product identifier
	host	Server host
	body_bytes_sent	Response bytes
	upstream_ response_time	Upstream response time
	http_user_agent	Client User-Agent

쇼핑몰 서비스의 access log는 일부 엔드포인트에 요청이 집중되고 HTTP 메서드, 상태 코드, 프로토콜, 콘텐츠 유형의 종류가 제한되는 특성을 가진다. 반면 사용자 에이전트, 클라이언트 주소, 질의 매개변수, 상품 식별자 등은 상대적으로 카디널리티가 높거나 디버깅 및 원본 추적 목적의 부가 정보로 사용된다. 이러한 특성은 저카디널리티와 고반복 필드 조합을 combo_id로 치환하고, 고카디널리티 또는 부가 필드는 extra_json으로 보존하는 HAL 구조의 평가에 적합하다. 다만 본 데이터셋은 웹 access log 중 하나의 서비스 유형에 기반하므로 다양한 로그 유형에 대한 일반화 가능성은 VI장에서 별도로 논의한다.

파이프라인의 동적 부하 조건은 Table 5와 같이 설정하였다. Logstash 워커 수는 노드의 vCPU 수와 동일한 2로 설정하여 CPU 자원을 최대한 활용하도록 하였다. 배치 크기는 Logstash 기본값인 125를 유지하였다. Java 가상 머신 힙 크기는 최소값과 최대값을 1 GB로 동일하게 고정하여, 실행 중 힙 크기 변화로 인한 측정 변동을 줄였다. Parquet 저장 시에는 실무에서 널리 사용되는 Snappy 압축 코덱을 사용하였다.

Table 5. Pipeline workload parameters

Parameter	Value
Logstash Workers	2
Batch Size	125
JVM Heap (-Xms / -Xmx)	1,007 MB (1g)
Parquet Codec	Snappy

HAL 전처리는 Logstash의Ruby filter 플러그인으로 구현하였다. 각 로그 레코드가 유입되면 조합 대상 필드 값을 추출하고 해시 맵으로 구성된 조합 테이블에서 해당 조합의 존재 여부를 조회한다. 매칭에 성공하면 정수형 조합 식별자를 기록하고 원본 조합 필드를 제거한다. 매칭에 실패하면 특수 식별자와 예외 값 배열을 사용하여 원본 값을 보존한다. 부가 필드는 별도의 레코드 구조로 재구성하였다. 구현은 약 100줄 이내의 스크립트로 구성되어 제안 방식이 복잡한 외부 시스템 없이 수집 계층 안에서 구현 가능함을 보여준다.

3. Variables and Measurement Methodology

본 실험의 독립변인은 두 가지이다. 첫 번째는 조합 테이블에 포함되는 필드 수이며 두 번째는 조합 테이블이 실제 로그 레코드를 얼마나 포괄하는지를 나타내는 매칭 비율이다. 필드 수에 따라 HAL200, HAL300, HAL400, HAL500의 네 가지 변형을 구성하였다. HAL200은 method와 endpoint의 2개 필드를 사용하고, HAL300은 status를 추가한다. HAL400은 protocol을 추가하며, HAL500은 content_type까지 포함하여 5개 필드 조합을 사용한다. 각 변형에 대해 조합 테이블의 등록 항목 수를 단계적으로 조정하여 다양한 매칭 비율을 구성하였다. 매칭 비율은 전체 로그 레코드 중 조합 테이블에 포함된 값 조합과 일치하는 레코드의 비율을 의미한다. 실험에서는 45.03%에서 100.00%까지의 구간을 구성하여 조합 테이블이 불완전한 경우와 모든 레코드를 포괄하는 경우를 모두 평가하였다. 구체적인 설정은 Table 6과 같다.

Table 6. Combo table configuration by HAL variant

Variant	Combo fields	Max table ID	Hit ratio range
HAL200	2 (method, endpoint)	3 / 4 / 5 / 7 / 10 / 19	53.55% ~ 100.00%
HAL300	3 (+status)	3 / 5 / 10 / 15 / 20 / 56	45.03% ~ 100.00%
HAL400	4 (+protocol)	3 / 5 / 10 / 15 / 20 / 64	45.03% ~ 100.00%
HAL500	5 (+content_type)	3 / 5 / 10 / 15 / 20 / 64	45.03% ~ 100.00%

실험의 기본 비교 기준은 HAL을 적용하지 않은 Avro-only 기준 구조이다. 기준 구조에서는 원본 JSON 로그가 기존 전처리를 거친 뒤, HAL 전처리 없이 동일한 Avro 직렬화 과정을 통해 Kafka에 적재된다. 따라서 HAL 적용 여부를 제외한 나머지 파이프라인 구성은 동일하게 유지하였다. 이하에서는 이 구조를 Avro-only로 표

기한다. Avro-only를 기본 기준 구조로 설정한 것은 HAL 적용 여부만을 독립적으로 비교하기 위한 것이다. Kafka compression과 같은 전송 계층 압축은 Avro-only와 HAL 양쪽 모두에 동일하게 결합될 수 있는 별도 변수이므로 이를 함께 변화시키면 HAL 전처리 자체의 기여도를 분리하기 어렵다. 따라서 본 실험에서는 Kafka 전송 계층 조건과 Parquet 저장 설정을 동일하게 유지하고, 직렬화 이전 의미 기반 전처리의 독립 효과를 우선 측정하였다.

종속변인은 전송 효율 지표와 자원 사용 지표로 구분하였다. 전송 효율 지표로는 Kafka 평균 메시지 크기, Kafka 토픽 적재 시간, 변환 시간, 총 소요 시간, 네트워크 대역폭 사용량, Parquet 저장 크기를 측정하였다. 자원 사용 지표로는 초당 메시지 처리량, CPU 사용률, Java 가상 머신 힙 사용률을 측정하였다. 각 지표의 측정 도구와 산출 방식은 Table 7과 같다.

Table 7. Measurement methods by dependent variable

Dependent variable	Measurement tool and method
Total / transform / load time	Lightweight shell script (sleep 5 + Kafka offset grep)
Peak MPS / Avg MPS	Timestamp-corrected MPS script (1s polling, divided by elapsed nanoseconds)
CPU usage	Logstash Node Stats API (process CPU field)
Kafka avg. message size	kafka-log-dirs.sh (topic size / message count)
MBPS	Kafka avg. message size × Avg MPS
Parquet size	S3 object storage size (direct query)
JVM heap usage	Logstash Node Stats API (/node/stats/jvm)

4. Experimental Procedure and Statistical Validation

측정 과정에서는 관측자 효과를 줄이기 위해 시간 지표와 자원 사용 지표를 분리하여 수집하였다. 총 소요 시간, 변환 시간, Kafka 적재 시간과 같은 시간 지표를 측정할 때에는 오버헤드가 작은 셸 스크립트만 사용하였다. 반면 CPU 사용률, 초당 메시지 처리량, 힙 사용률과 같은 자원 지표는 1초 간격의 별도 수집 스크립트를 사용하여 독립 실행에서 측정하였다. 이는 측정 도구 자체가 CPU나 입출력 자원을 사용하여 실험 결과에 영향을 주는 것을 줄이기 위한 절차이다.

각 실험 조건은 기본적으로 5회 반복 수행하였고 실행 간 변동이 큰 조건에서는 반복 횟수를 10회로 확대하여 평균과 표준편차의 안정성을 높였다. 측정값 중 시스템 로그와 대조했을 때 일시적인 프로세스 오류나 외부 요인으로 판단되는 값은 제외하고, 나머지 값에 대해 평균과 표준편차를 산출하였다. 이러한 반복 측정은 단일 실행 결과에 의존하지 않고 조건 간 차이를 안정적으로 비교하기 위한 것이다.

통계적 검증에는 독립 표본 t 검정을 사용하였다. 유의수준은 0.05로 설정하였다. 메시지 크기, 적재 시간, 네트워크 대역폭과 같이 성능 개선을 주장하는 지표에 대해서는 기준 구조와 HAL 적용 구조 간 차이가 통계적으로 유의한지 확인하였다. 반면 총 소요 시간과 처리량의 경우에는 HAL 적용으로 인한 성능 저하가 관찰되는지를 확인하는 데 초점을 두었다. 비유의성은 엄밀한 의미의 동등성 검정을 대체하지는 않으나 본 실험 범위에서 처리 성능 저하의 근거가 확인되지 않았음을 판단하는 보조 근거로 사용하였다.

V. Results and Analysis

본 장에서는 4장에서 설계한 실험의 결과를 제시하고, HAL이 전송 효율, 저장 효율, 처리 비용에 미치는 영향을 분석한다. 전체 결과는 세 가지 경향으로 요약된다. 첫째, 조합 테이블에 포함되는 필드 수가 증가할수록 Kafka 메시지 크기와 적재 시간의 절감 폭이 커진다. 둘째, 동일한 HAL 변형에서는 매칭 비율이 높아질수록 전송 효율이 개선된다. 셋째, HAL 전처리는 변환 단계에서 추가 연산을 요구하지만, 충분한 필드 수와 매칭 비율이 확보되면 적재 시간 절감 효과가 이를 상쇄하여 전체 처리 시간과 처리량의 유의한 저하가 관찰되지 않는다. 각 절에서는 분석 목적에 맞는 시각적 비교(Fig. 4~6)를 제시하며 HAL 전 변형 및 전 매칭 비율에 대한 전체 벤치마크 수치는 부록 Table A1에 수록하였다. 이하 절에서 이를 세부 지표별로 분석한다.

1. Transmission Efficiency by Combo Field Count

조합 테이블에 포함되는 필드 수가 전송 효율에 미치는 영향을 분석하기 위해 모든 HAL 변형의 매칭 비율을 100%로 고정한 조건에서 HAL을 적용하지 않은 Avro-only 기준 구조와 비교하였다. 이 조건은 각 변형이 가질 수 있는 최대 절감 효과를 평가하기 위한 것이다. Fig. 4는 이러한 결과를 시각적으로 정리한 것이다.

Kafka 평균 메시지 크기는 Fig. 4(a)에서 확인할 수 있듯이 조합 필드 수가 증가함에 따라 전반적으로 감소하는

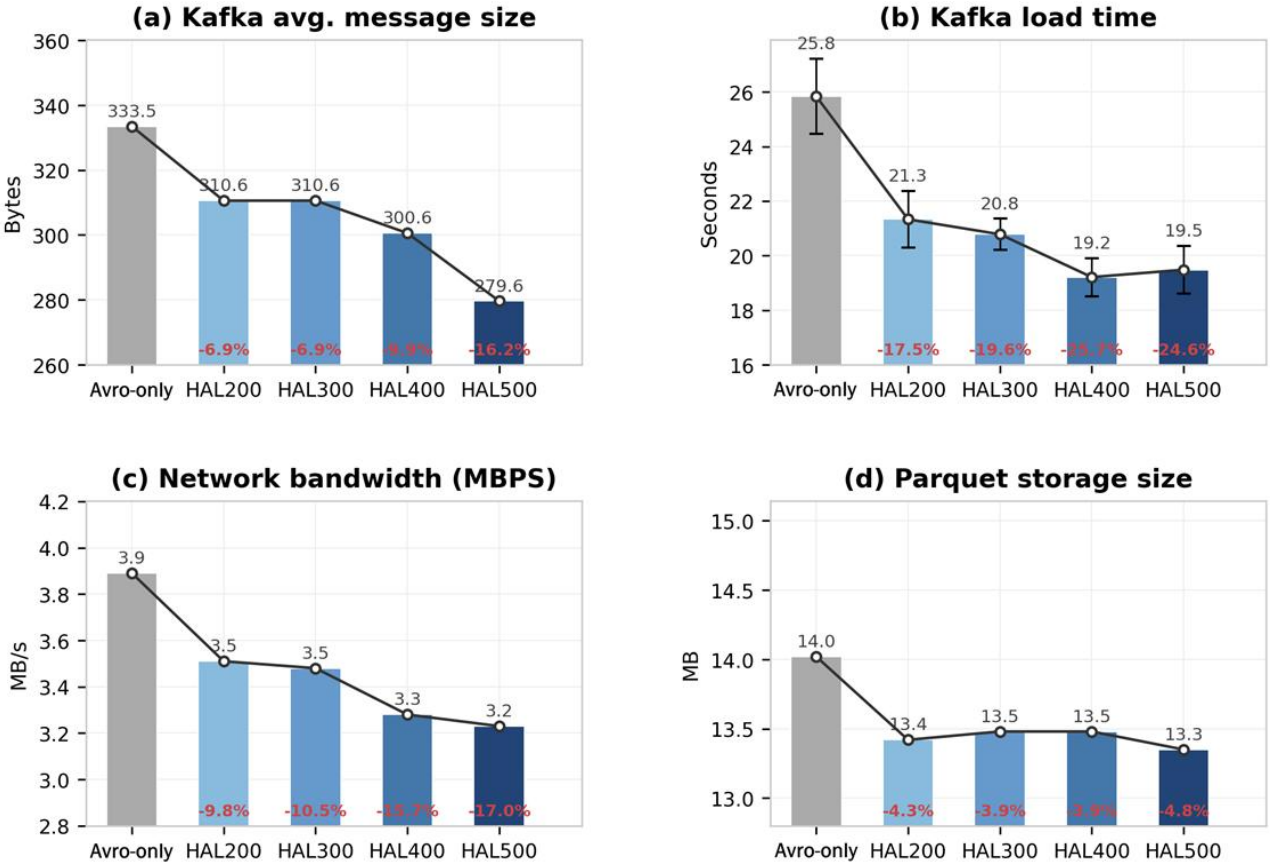

Fig. 4. Transmission efficiency by combo field count (hit ratio = 100%)

경향을 보인다. 기준 구조의 평균 메시지 크기는 333.46바이트였으며, HAL200은 310.57바이트로 6.9% 감소하였다. HAL500은 279.60바이트로 감소하여 기준 구조 대비 16.2%의 절감률을 보였다. 이는 조합 테이블에 포함되는 필드가 많아질수록 하나의 정수형 식별자로 대체되는 문자열의 총 길이가 증가하기 때문이다. 특히 HAL200과 HAL300의 메시지 크기가 거의 동일하게 나타난 점은 Fig. 4(a)에서도 확인할 수 있다. HAL300은 HAL200에 상태 코드 필드를 추가하지만, 상태 코드는 대체로 세 자리의 짧은 문자열로 구성되므로 절감되는 바이트 수가 제한적이다. 반면 protocol과 content_type과 같이 상대적으로 긴 문자열 필드를 포함하는 HAL400과 HAL500에서는 절감 효과가 더 크게 나타난다.

Kafka 토픽 적재 시간은 Fig. 4(b)와 같이 메시지 크기 감소와 함께 줄어드는 경향을 보인다. 기준 구조의 평균 적재 시간은 25.84초였으며, HAL400은 19.21초, HAL500은 19.48초로 측정되었다. 이는 각각 25.7%와 24.6%의 절감에 해당하며, 두 조건 모두 기준 구조 대비 통계적으로 유의한 개선을 보였다. HAL400의 적재 시간이 HAL500보다 약간 짧게 나타났으나 두 조건 간 차이는 실험 반복에서 발생하는 변동 범위 내에 있어 통계적으로 유의하지 않은 것으로 해석된다.

네트워크 대역폭 사용량은 Fig. 4(c)에서와 같이 메시지 크기 감소에 따라 함께 감소하는 경향을 보인다. HAL500은 기준 구조 대비 17.0%의 대역폭 절감을 달성하였다. 이는 HAL이 저장 크기뿐 아니라 Kafka로 전송되는 데이터 양 자체를 감소시킨다는 점을 보여준다. 네트워크 대역폭은 클라우드 환경에서 비용과 성능에 직접적인 영향을 미치는 자원이므로 이러한 절감은 실무적으로 중요한 의미를 가진다.

Parquet 저장 크기 역시 Fig. 4(d)에서 확인할 수 있듯이 추가적인 감소를 보인다. 기준 구조의 저장 크기는 14.02MB였으며 HAL500은 13.35MB로 감소하여 4.8%의 절감률을 나타냈다. 절감 폭이 Kafka 메시지 크기보다 상대적으로 작은 이유는 Parquet이 자체적으로 컬럼 기반 압축과 사전 인코딩을 수행하기 때문이다. 그럼에도 불구하고 HAL 적용 후 저장 크기가 증가하지 않고 감소한 것은 제안 방식이 전송 단계뿐 아니라 저장 단계에서도 부정적인 영향을 주지 않음을 의미한다.

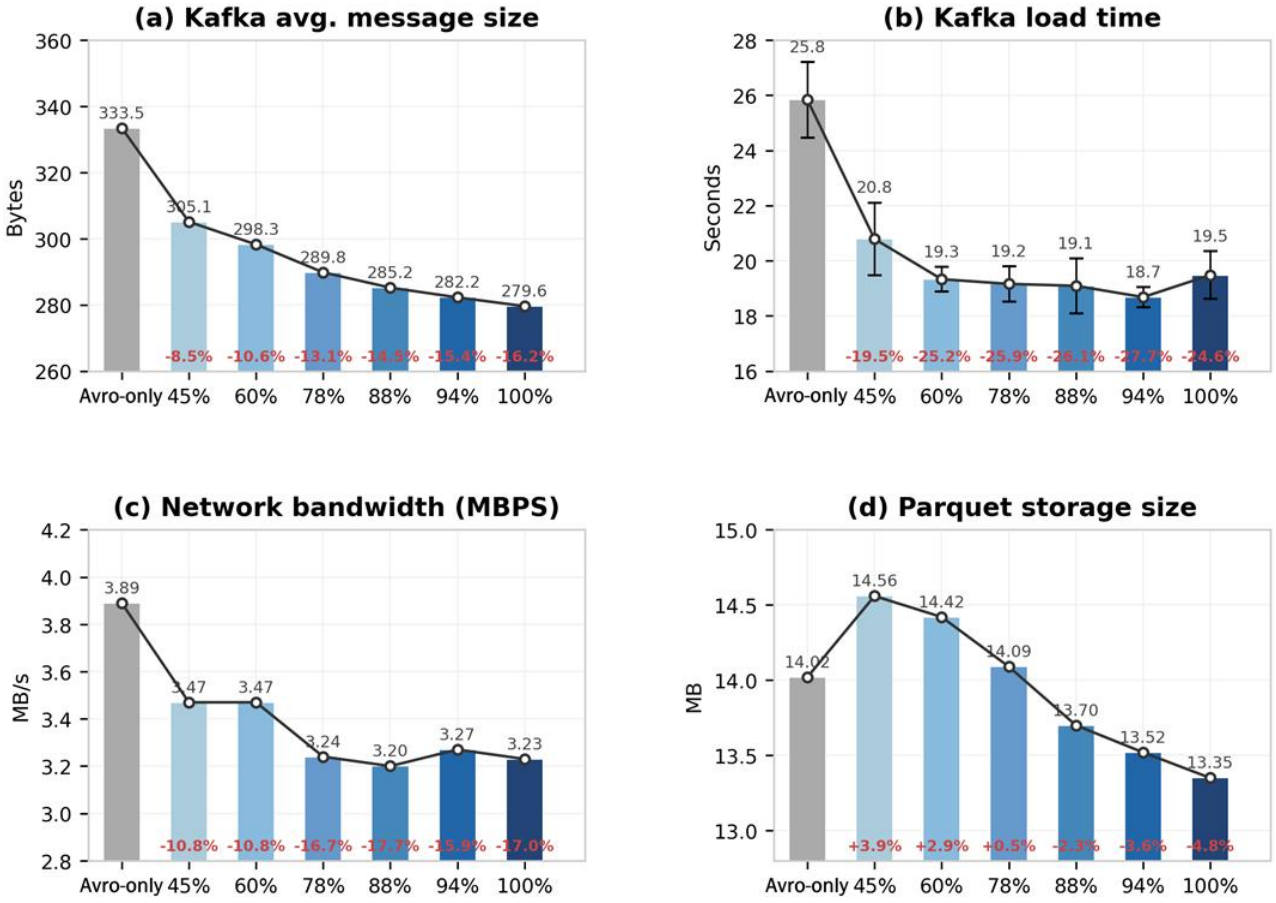

Fig. 5. Performance variation by hit ratio (HAL500)

2. Performance Variation by Hit Ratio

매칭 비율이 성능에 미치는 영향을 분석하기 위해 절감 효과가 가장 크게 나타난 HAL500을 기준으로 매칭 비율을 45.03%에서 100.00%까지 변화시키며 Avro-only 기준 구조와 비교하였다. Fig. 5는 이러한 결과를 시각적으로 나타낸 것이다.

Kafka 평균 메시지 크기는 Fig. 5(a)에서 확인할 수 있듯이 매칭 비율이 증가할수록 감소하는 경향을 보인다. 매칭 비율이 45.03%인 조건에서도 평균 메시지 크기는 기준 구조 대비 8.5% 감소하였으며 매칭 비율이 100%에 도달하면 절감률은 16.2%까지 증가하였다. 이는 조합 테이블과 일치하는 레코드의 비중이 증가함에 따라 원본 문자열 대신 정수형 식별자로 치환되는 레코드가 많아지기 때문이다.

Kafka 적재 시간 역시 Fig. 5(b)에서와 같이 매칭 비율 증가에 따라 전반적으로 감소하는 경향을 보인다. 특히 매칭 비율이 45.03%인 조건에서도 적재 시간은 기준 구조 대비 19.5% 감소하였다. 이는 일부 레코드가 예외 값 배열을 사용하더라도 나머지 레코드에서의 메시지 크기 감소 효과가 전체 적재 시간 개선으로 이어질 수 있음을 의미한다. 따라서 HAL은 조합 테이블이 완전하지 않은 초기 운영 단계에서도 전송 효율 개선 효과를 제공할 수 있다.

네트워크 대역폭 사용량은 Fig. 5(c)에서와 같이 메시지 크기 변화와 유사한 경향을 보이며 매칭 비율 증가에 따라 감소한다. 이는 전송되는 데이터의 크기 자체가 줄어들기 때문에 나타나는 자연스러운 결과이다.

반면 Parquet 저장 크기는 Fig. 5(d)에서 확인할 수 있듯이 Kafka 메시지 크기와 달리 비선형적인 변화를 보인다. HAL500의 매칭 비율이 45.03%에서 77.84%인 구간에서는 Parquet 저장 크기가 기준 구조인 14.02MB보다 오히려 증가하였으며, 특히 45.03% 조건에서는 14.56MB로 가장 크게 나타났다. 이는 Kafka 메시지 크기와 Parquet 저장 크기가 서로 다른 저장 원리에 의해 결정되기 때문이다. Kafka 메시지 크기는 레코드 단위 Avro payload의 바이트 수에 직접 영향을 받으므로 일부 레코드만 매칭되어도 감소할 수 있다. 반면 Parquet 저장 크기는 컬럼 단위 사전 인코딩, 반복 길이 인코딩, 그리고 배열 구조의 repetition/definition level 표현에 영향을 받는다. 매칭 실패 레코드가 많을 경우 ovr 배열에 원본 문자열 조합이 저장되며, 이 가변 길이 배열과 혼합된 값 분포가 Parquet의 컬럼 압축 효율을 저하시킬 수 있다. 그

러나 매칭 비율이 87.92% 이상으로 높아지면 ovr 배열이 대부분 비어 있고 combo_id의 반복성이 증가하므로, 저장 크기는 기준 구조 이하로 감소하기 시작한다.

이러한 결과는 조합 테이블 설계에서 매칭 비율이 중요한 기준임을 보여준다. 특히 전송 효율뿐 아니라 저장 효율까지 함께 고려할 경우, 단순히 많은 필드를 포함하는 것보다 높은 매칭 비율을 유지할 수 있는 필드 조합을 선택하는 것이 중요하다. 실제로 매칭 비율이 낮은 조건에서도 메시지 크기와 적재 시간에서 약 10% 내외의 개선이 확인되었으며 매칭 비율이 증가할수록 메시지 크기, 네트워크 사용량, 적재 시간에서 약 15~25% 수준의 절감 효과가 나타났다. 이러한 결과는 HAL이 부분적인 매칭 환경에서도 안정적인 성능 개선을 제공하며, 충분한 매칭 비율이 확보될 경우 전송과 저장 효율을 동시에 효과적으로 개선할 수 있음을 보여준다.

3. Transform Cost and Resource Usage

HAL은 직렬화 이전 단계에서 추가 변환을 수행하므로 일정한 변환 비용이 발생한다. Fig. 6은 매칭 비율 100% 조건에서 Avro-only 기준 구조와 각 HAL 변형의 변환 시간과 적재 시간 구성을 비교한 결과를 보여준다.

Fig. 6에서 확인할 수 있듯이 HAL 적용 시 변환 시간은 기준 구조 대비 약 16%에서 21% 증가하였으나 HAL200부터 HAL500까지 증가 폭에는 큰 차이가 나타나지 않았다. 조합 필드 수가 증가함에도 변환 시간은 약 47초에서 49초 범위에 머물렀으며 이는 추가 비용의 대부분이 해시 맵 조회 자체가 아닌 Logstash 필터 실행 과정에서 발생하는 고정 비용에 기인함을 의미한다. 반면 적재 시간은 메시지 크기 감소에 따라 감소하며 변환 비용 증가를 상당 부분 상쇄한다. HAL200과 HAL300에서는 적재 시간 감소가 변환 비용 증가를 완전히 상쇄하지 못해 총 소요 시간이 증가하였으나 HAL400에서는 그 차이가 크게 줄어들고 HAL500에서는 총 소요 시간이 기준 구조와 거의 동일한 수준을 유지하였다. 실제로 기준 구조의 총 소요 시간은 66.62초, HAL500은 66.93초로 나타났으며 두 조건 간 차이는 통계적으로 유의하지 않았다.

이러한 결과는 HAL이 단순히 추가 비용을 발생시키는 방식이 아니라 CPU 연산 일부를 선투자하여 전송 비용을 줄이는 구조로 동작함을 보여준다. 기준 구조에서는 CPU 자원이 주로 대용량 페이로드의 직렬화와 전송에 사용되는 반면 HAL 적용 시에는 일부 연산이 필드 변환에 사용되지만, 그 결과 전송 데이터 크기가 감소하여 전체 비용이 재분배된다.

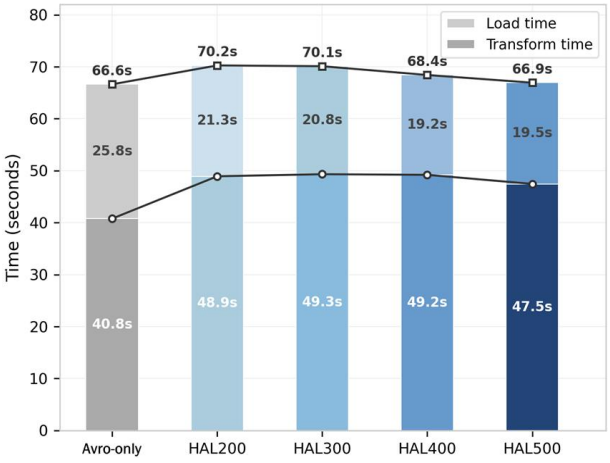

Fig. 6. Time composition by HAL variant (hit ratio = 100%)

CPU 사용률은 기준 구조와 HAL500 모두 약 92~93%로 나타나 실험 환경이 CPU 포화 상태였음을 보여준다. 이러한 조건에서는 총 처리 시간을 크게 단축하기 어렵지만 HAL은 동일한 CPU 자원 내에서 연산의 구성을 재조정하여 전송 효율을 개선한다. 메모리 사용 측면에서도 추가 부담은 제한적이었다. JVM 힙 사용률은 두 조건에서 유사한 범위를 보였으며 이는 HAL이 해시 맵 조회와 배열 처리 중심의 경량 연산으로 구성되어 별도의 대규모 메모리를 요구하지 않기 때문이다.

HAL의 효과는 조합 필드 수와 매칭 비율에 의해 결정된다. 전송 효율은 낮은 매칭 비율에서도 개선되지만 변환 비용을 상쇄하고 저장 효율까지 확보하기 위해서는 충분한 문자열 절감이 가능한 필드 조합이 필요하다. 본 실험에서는 4개 이상의 필드를 포함하는 경우부터 총 소요 시간 증가가 제한되었으며 5개 필드 조건에서 처리량 저하 없이 가장 높은 전송 효율이 달성되었다. 특히 HAL500 조건에서는 메시지 크기 약 16%, 네트워크 대역폭 약 17%, 적재 시간 약 24%의 감소를 보이면서도 총 처리 시간과 처리량에는 유의한 변화가 없었으며 이를 통해 전송 효율과 처리 성능을 동시에 개선할 수 있음을 확인하였다.

VI. Conclusion

본 논문은 Kafka 기반 로그 수집 파이프라인에서 Avro 직렬화 이후에도 값 수준의 반복이 남아 전송 비용을 증가시키는 문제에 주목하였다. 이를 해결하기 위해 JSON 구조화 로그의 반복 필드 조합을 직렬화 이전 단계에서 정수형 식별자로 변환하는 HAL을 제안하였다. HAL은 기존 Kafka 브로커와 저장 구조를 변경하지 않고 전처리 단계

에서 적용 가능하며, 사전에 정의되지 않은 값 조합도 배열 형태로 보존하여 데이터 손실 없이 복원할 수 있도록 설계되었다.

Nginx 액세스 로그 849,000건을 대상으로 한 실험 결과, HAL은 메시지 크기, 네트워크 대역폭, 적재 시간에서 약 16~24%의 절감 효과를 보였고, Parquet 저장 크기에서도 추가 감소가 확인되었다. 변환 과정에서 연산 비용은 발생하였으나, 충분한 필드 조합을 사용할 경우 적재 시간 절감 효과가 이를 대부분 상쇄하였다. 그 결과 총 소요 시간과 처리량에서 기준 구조 대비 통계적으로 유의한 성능 저하의 근거는 관찰되지 않았다. 다만 이는 엄밀한 동등성 검정을 의미하지 않으므로, 본 연구는 차이의 크기와 신뢰구간을 함께 제시하여 성능 영향이 실무적으로 작은 범위에 있음을 확인하였다.

본 연구의 공학적 의의는 직렬화 이후의 범용 압축이 아니라 직렬화 이전 단계의 데이터 구조 단순화를 통해 전송 및 저장 효율을 개선할 수 있음을 실증적으로 보였다는 점이다. HAL은 기존 직렬화 및 저장 기술을 대체하기보다 입력 데이터를 사전에 정돈하여 후속 처리 부담을 줄이는 보완적 접근으로 해석될 수 있다.

향후 연구에서는 다중 브로커와 다중 파티션 기반의 분산 Kafka 환경에서 HAL의 효과를 검증할 필요가 있다. 또한 본 연구가 Nginx access log 기반 웹 요청 로그를 대상으로 하였으므로, 시스템 로그, 애플리케이션 이벤트 로그, 보안 로그 등 반복성과 카디널리티 특성이 다른 로그 유형에 대한 일반화 가능성도 추가로 분석해야 한다. 반복 필드가 많은 로그에서는 HAL의 효과가 기대되지만, 자유 형식 메시지나 IP 주소, URL, 사용자 식별자처럼 고카디널리티 값이 지배적인 경우 효과가 제한될 수 있다. 따라서 로그 유형별 필드 반복성과 조합 안정성을 함께 고려해야 한다. 나아가 조합 테이블의 자동 생성 및 갱신, Kafka compression과의 병행 적용, Logstash 기반 구현의 변환 오버헤드를 줄이기 위한 경량 실행 구조도 중요한 후속 과제로 고려될 수 있다.

ACKNOWLEDGEMENT

This research was supported by the MSIT(Ministry of Science, ICT), Korea, under the National Program for Excellence in SW, supervised by the IITP(Institute for Information communications Technology Planning&Evaluation) in 2026(2021-0-01440)

REFERENCES

[1] B. Li, X. Peng, Q. Xiang, H. Wang, T. Xie, J. Sun, and X. Liu, "Enjoy Your Observability: An Industrial Survey of Microservice Tracing and Analysis," Empirical Software Engineering, Vol. 27, No. 1, Article 25, Jan. 2022. DOI: 10.1007/s10664-021-10063-9

[2] M. A. Uddin, S. Weerasinghe, D. Gajewski, M. Akbarsharifi, R. Akbarsharifi, C. Stoner, T. Cerny, and S. He, "Microservice Logs Analysis Employing AI: A Systematic Literature Review," Journal of Systems and Software, Vol. 236, 112786, June 2026. DOI: 10.1016/j.jss.2026.112786

[3] M. Usman, S. Ferlin, A. Brunstrom, and J. Taheri, "A Survey on Observability of Distributed Edge & Container-Based Microservices," IEEE Access, Vol. 10, pp. 86904-86919, July 2022. DOI: 10.1109/ACCESS.2022.3193102

[4] U. Faseeha, H. J. Syed, F. Samad, S. Zehra, and H. Ahmed, "Observability in Microservices: An In-Depth Exploration of Frameworks, Challenges, and Deployment Paradigms," IEEE Access, Vol. 13, pp. 72011-72039, Apr. 2025. DOI: 10.1109/ACCESS.2025.3562125

[5] J. Kreps, N. Narkhede, and J. Rao, "Kafka: a Distributed Messaging System for Log Processing," Proc. of the 6th International Workshop on Networking Meets Databases (NetDB), pp. 1-7, Athens, Greece, June 2011.

[6] M. Kleppmann, "*Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*," O'Reilly Media, pp. 116-127, 2017.

[7] P. Dobbelaere and K. S. Esmaili, "Kafka versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations," Proc. of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS '17), pp. 227-238, Barcelona, Spain, June 2017. DOI: 10.1145/3093742.3093908

[8] Apache Software Foundation, Apache Avro Specification (Version 1.11.1), *https://avro.apache.org/docs/1.11.1/specification/*

[9] S. Henning and W. Hasselbring, "Benchmarking Scalability of Stream Processing Frameworks Deployed as Microservices in the Cloud," Journal of Systems and Software, Vol. 208, 111879, Feb. 2024. DOI: 10.1016/j.jss.2023.111879

[10] K. J. Matteussi, J. C. S. dos Anjos, V. R. Q. Leithardt, and C. F. R. Geyer, "Performance Evaluation Analysis of Spark Streaming Backpressure for Data-Intensive Pipelines," Sensors, Vol. 22, No. 13, 4756, June 2022. DOI: 10.3390/s22134756

[11] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive Analysis of Web-Scale Datasets," Proceedings of the VLDB Endowment, Vol. 3, No. 1-2, pp. 330-339, Sept. 2010. DOI: 10.14778/1920841.1920886

[12] W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, and E. M. Nguifo, "An Experimental Survey on Big Data Frameworks," Future

Generation Computer Systems, Vol. 86, pp. 546-564, Sept. 2018. DOI: 10.1016/j.future.2018.04.032

[13] Apache Software Foundation, Apache Kafka Documentation: Producer Configs (compression.type), https://kafka.apache.org/documentation/#producerconfigs_compression.type

[14] J. Liu, J. Zhu, S. He, P. He, Z. Zheng, and M. R. Lyu, "Logzip: Extracting Hidden Structures via Iterative Clustering for Log Compression," Proc. of 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 863-873, San Diego, USA, Nov. 2019. DOI: 10.1109/ASE.2019.00085

[15] J. Wei, G. Zhang, Y. Wang, Z. Liu, Z. Zhu, J. Chen, T. Sun, and Q. Zhou, "On the Feasibility of Parser-based Log Compression in Large-Scale Cloud Systems," Proc. of the 19th USENIX Conference on File and Storage Technologies (FAST '21), pp. 249-262, 2021.

[16] X. Li, H. Zhang, V.-H. Le, and P. Chen, "LogShrink: Effective Log Compression by Leveraging Commonality and Variability of Log Data," Proc. of 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE), pp. 243-254, Lisbon, Portugal, Apr. 2024. DOI: 10.1145/3597503.3608129

[17] S. Shan, Y. Huo, H. Zhong, Z. Wang, Y. Su, and Z. Zheng, "LogFold: Compressing Logs with Structured Tokens and Hybrid Encoding," Proc. of 2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE), Rio de Janeiro, Brazil, Apr. 2026. (to appear)

[18] Y. Liu, K. Zhang, Z. Chen, and Z. Zheng, "LogPrism: Unifying Structure and Variable Encoding for Effective Log Compression," arXiv preprint, *https://arxiv.org/abs/2601.17482*, Jan. 2026.

[19] K. Yao, M. Sayagh, W. Shang, and A. E. Hassan, "Improving State-of-the-Art Compression Techniques for Log Management Tools," IEEE Transactions on Software Engineering, Vol. 48, No. 8, pp. 2748-2760, Aug. 2022. DOI: 10.1109/TSE.2021.3069958

[20] S. Yu, Y. Wu, Y. Li, and P. He, "Unlocking the Power of Numbers: Log Compression via Numeric Token Parsing," Proc. of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), pp. 919-930, Sacramento, USA, Oct. 2024. DOI: 10.1145/3691620.3695474

[21] H. Lin, J. Zhou, B. Yao, M. Guo, and J. Li, "Cowic: A Column-Wise Independent Compression for Log Stream Analysis," Proc. of 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 21-30, Shenzhen, China, May 2015. DOI: 10.1109/CCGrid.2015.45

[22] B. Tang, S. Yang, Z. Shen, W. Zhang, X. Lin, and Z. Tian, "LogLite: Lightweight Plug-and-Play Streaming Log Compression," Proceedings of the VLDB Endowment, Vol. 18, No. 11, pp. 3757-3770, 2025. DOI: 10.14778/3749646.3749652

Authors

Min-gi Lee is currently an undergraduate student in the Division of Computer Science and Engineering at Sahmyook University, Korea, and in his fourth year of the B.S. program.

Min-gi Lee has been studying as an undergraduate researcher at the Cloud Lab in the Division of Computer Science and Engineering at Sahmyook University, Korea, since 2025. His research interests include Cloud Computing and Computer Architecture.

Choong-hee Cho received the B.S. degree in Computer Engineering from Sahmyook University, Korea, in 2010, and the Ph.D. degree from the Korea University of Science and Technology, Korea, in 2019.

Dr. Cho joined the faculty of the Department of Data Cloud Engineering at Sahmyook University, Seoul, Korea, in 2022. He is currently an Assistant Professor in the Department of Data Cloud Engineering, Sahmyook University. His research interests include cloud architecture and microservices architecture (MSA).

APPENDIX

Table A1. Full benchmark results by HAL variant and hit ratio

Variant	Hit (%)	Kafka (B)	Load (s)	Transform (s)	Total (s)	MBPS (MB/s)	Parquet (MB)
Avro-only	–	333.46	25.84±1.38	40.78±1.39	66.62±1.53	3.89	14.02
HAL200	53.55	317.81	22.87±1.60	47.42±1.18	70.29±1.98	3.52	13.94
	63.60	316.32	23.56±2.11	49.17±1.82	72.73±1.72	3.43	13.88
	71.60	314.97	22.57±1.37	49.67±1.18	72.24±2.07	3.45	13.75
	82.60	313.44	22.39±1.32	48.02±0.98	70.41±1.37	3.52	13.66
	91.38	312.05	21.51±1.36	48.32±2.11	69.83±2.29	3.48	13.55
	100.00	310.57	21.33±1.03	48.92±1.83	70.26±2.07	3.51	13.42
HAL300	45.03	320.88	23.25±1.73	50.71±1.54	73.96±2.03	3.57	14.41
	59.84	317.93	21.65±1.33	49.88±1.04	71.53±1.45	3.42	14.27
	77.84	314.98	22.42±1.23	50.27±0.61	72.69±1.35	3.32	13.93
	87.92	312.97	21.32±0.44	49.31±1.54	70.63±1.58	3.43	13.69
	94.30	311.72	21.35±1.18	49.21±1.62	70.56±1.64	3.46	13.62
	100.00	310.59	20.78±0.57	49.32±1.30	70.11±1.64	3.48	13.48
HAL400	45.03	315.85	22.75±1.61	49.68±1.20	72.43±1.89	3.47	14.41
	59.84	311.54	20.78±1.72	49.62±1.45	70.41±2.21	3.49	14.29
	77.84	306.96	20.84±1.26	49.58±1.22	70.42±1.27	3.50	14.01
	87.92	304.05	19.70±0.82	49.94±1.09	69.64±1.11	3.36	13.72
	94.30	302.27	20.17±1.00	48.68±1.02	68.85±0.53	3.46	13.57
	100.00	300.56	19.21±0.70	49.20±0.71	68.41±0.72	3.28	13.48
HAL500	45.03	305.10	20.79±1.32	48.69±1.04	69.48±1.66	3.47	14.56
	59.84	298.26	19.33±0.45	49.32±0.85	68.65±1.06	3.47	14.42
	77.84	289.82	19.16±0.64	48.81±0.82	67.97±1.37	3.24	14.09
	87.92	285.20	19.09±1.00	48.96±0.92	68.04±0.91	3.20	13.70
	94.30	282.25	18.68±0.36	48.02±2.61	66.70±2.40	3.27	13.52
	100.00	279.60	19.48±0.87	47.45±0.72	66.93±0.55	3.23	13.35